Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information and Computation 190 (2004) 100–116

Information and Computation

www.elsevier.com/locate/ic

The submatrices character count problem: an efficient solution using separable values☆

Amihood Amir,[a,b,*,1] Kenneth W. Church,[c,2] and Emanuel Dar[a,3]

[a] *Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel*
[b] *Georgia Tech, USA*
[c] *AT&T Labs – Research, Shannon Laboratory, D235, 180 Park Avenue, Florham Park, NJ 07932-0971, USA*

Received 29 November 2001; revised 3 December 2003

Abstract

The *subsequence character count problem* has as its input an array $S = S[1], \ldots, S[n]$ of symbols over alphabet Σ and a natural number m. Its output is: for every i, $i = 1, \ldots, n - m + 1$, the number of *different* alphabet symbols occurring in the subsequence $S[i], S[i+1], \ldots, S[i+m-1]$. The subsequence character count problem is a natural problem that has many uses. It can be solved in linear time for finite alphabets and in time $O(n \log m)$ for infinite alphabets. When the character count problem is generalized to two dimensions it becomes the *submatrix character count problem*. Its input is an $n \times n$ matrix T over alphabet Σ and a natural number m. Its output is: for every i, j, $i, j = 1, \ldots, n - m + 1$, the number of *different* alphabet symbols occurring in the submatrix $T[i+k, j+\ell]$, $k = 0, \ldots, m-1$; $\ell = 0, \ldots, m-1$. The straightforward one-dimensional solution slides a window along the text adding an element and deleting an element at every step. The problem with two dimensions is that at every move of the window there are m elements added and m deleted. In this paper, we present an alternate one-dimensional solution that generalizes to two dimensions. We achieve a $O(n^2)$ time solution to the submatrix character count problem over a finite alphabet and a $O(n^2 \log m)$ solution over an infinite alphabet.

☆ A short abstract of the results presented in this paper appeared in the Proceedings of the 13th Annual ACM/SIAM Symposium on Discrete Algorithms [3].

* Corresponding author. Fax: +972-3-736-0498.
E-mail addresses: amir@macs.biu.ac.il (A. Amir), kwc@research.att.com (K.W. Church), dar@cs.biu.ac.il (E. Dar).

[1] Partially supported by NSF Grant CCR-01-04494 and ISF Grant 282/01. Part of this work was done while the author was at AT&T Labs – Research and DIMACS.

[2] Fax: 1-973-360-8077.

[3] This work is part of E. Dar's Ph.D. Thesis.

0890-5401/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2003.12.001

1. Introduction

One of the tasks of algorithms design is computing a value of every fixed-sized subarray in a given array. For example, one may want to compute the sum of every m consecutive elements in a list of numbers.

A basic and very effective technique for solving such problems is the "sliding window" technique, taught in introductory programming courses. The idea is to consider a certain value within a consecutive subarray, and then maintain that value as the "window" slides by one location. As an example consider the above-mentioned *subsequence sums* problem, formally defined as follows.

Definition 1. The *subsequence sums* problem is:
INPUT: Array $S = S[1], \ldots, S[n]$ of numbers and a natural number m.
OUTPUT: Array $U = U[1], \ldots, U[n]$ such that for every i, $i = 1, \ldots, n - m + 1$, $U[i] = \sum_{j=i}^{i+m-1} S[j]$.

This problem can be solved trivially by the "sliding window" technique in time $O(n)$. Simply take the sum of the first m numbers, $\sum_{i=1}^{m} S[i]$, as the solution for $i = 1$. Subsequently compute the solution to $i + 1$ by subtracting $S[i]$ and adding $S[i + m]$ to the solution of i.

A trifle more challenging is the *subsequence character count problem.*

Definition 2. The *subsequence character count* problem is defined as follows:
INPUT: Array $S = S[1], \ldots, S[n]$ of symbols over alphabet Σ and a natural number m.
OUTPUT: Array $U = U[1], \ldots, U[n]$ such that for every i, $i = 1, \ldots, n - m + 1$, $U[i] =$ the number of *different* alphabet symbols occurring in the subsequence $S[i], S[i + 1], \ldots, S[i + m - 1]$.

In [5] the problem was used to solve the *parameterized matching* problem. The "sliding window" technique keeps a list of counters of the number of occurrences of every symbol in the window, and a variable D designating the number of different symbols in the window. When sliding from position i to position $i + 1$, subtract 1 from the counter of the symbol in location i. If the counter drops to 0 then subtract 1 from D. Add 1 to the counter of the symbol in location $i + m$ and, if it changed from 0 to 1 then add 1 to D.

The time analysis depends on the complexity of accessing the counters in the list. For alphabet $\{1, \ldots, n\}$ this can clearly be done in time $O(n)$. It was also shown that in the comparison model the problem is solved in time $\Theta(n \log m)$ if the alphabet is ordered. For unordered alphabets the problem requires at least time $\Omega(nm)$. The lower bounds were achieved via reduction from the element distinctness problem [13].

While the "sliding window" works simply and efficiently for subsequences of vectors, it is sometimes necessary to apply it to two-dimensional arrays. Of particular interest to us was the *submatrix character count* problem, which is the two-dimensional version of the subsequence character count problem. The submatrix character count problem was motivated by indexing color images. The number of different colors in a subarea of an image can be considered a "signature". Two images may be considered similar if the color signatures are similar in respective subareas. There are many image processing tools that use color histograms as a measure (see, e.g., [10]). A simplified version of the above idea is comparing the color signature of respective $m \times m$ submatrices of two images.

We formally define the two-dimensional generalizations of the subsequence sums and character count problems.

Definition 3.
(a) The *submatrix sum* problem is:
INPUT: Two-dimensional $n \times n$ array T of numbers and a natural number m.
OUTPUT: Two-dimensional $n \times n$ array U such that for every $i, j \in \{1, \ldots, n-m+1\}$, $U[i,j] = \sum_{k=i}^{i+m-1} \sum_{\ell=j}^{j+m-1} T[k,\ell]$.
(b) The *submatrix character count* problem is:
INPUT: Two-dimensional $n \times n$ array T of symbols over alphabet Σ and a natural number m.
OUTPUT: Two-dimensional $n \times n$ array U such that for every $i, j \in \{1, \ldots, n-m+1\}$, $U[i,j] =$ the number of *different* alphabet symbols occurring in the submatrix $T[k,\ell]$, $k = i, \ldots, i+m-1$, $\ell = j, \ldots, j+m-1$.

A straightforward generalization of the "sliding window" idea will not work since a shift from position $[i,j]$ to, say $[i, j+1]$ in the text, requires m operations to keep the window property satisfied.

The submatrix character count problem is a special case of a *color range query*.

Definition 4. The *color range query* problem is:
PREPROCESS: Two-dimensional $n \times n$ array T of symbols over alphabet Σ – the colors.
Subsequently we are interested in answers to:
QUERY: Given intervals $[i_1, j_1]$ and $[i_2, j_2]$, $i_1, i_2, j_1, j_2 \in \{1, \ldots, n\}$ and $i_1 \leqslant j_1$, $i_2 \leqslant j_2$ give the number of *different* alphabet symbols (colors) occurring in the submatrix $T[k,\ell]$, $k = i_1, \ldots, j_1$, $\ell = i_2, \ldots, j_2$.

Janardan and Lopez [11] showed that with a $O(n^2 \log^2 n)$ preprocessing one can answer queries in time $O(\log^2 n)$. This means that the submatrices character count problem can be solved in time $O(n^2 \log^2 n)$ by preprocessing and then querying, for every location, the $m \times m$ submatrix starting at that location.

We are not aware of a faster direct approach for solving the submatrix character count problem. However, problems with a similar flavor, where the desired calculation is a convolution, are solved in electrical engineering by a method called *Separable Convolutions* or *Separable Filters* [9]. A similar notion was used by Bird [7] and Baker [6] to solve the two-dimensional pattern matching problem.

The contributions of this paper are twofold.
1. Inspired by the engineering notion of *separable convolutions* we formalize the notion of *separable values*. We think it is important to keep this method in mind when considering problems on two-dimensional arrays. The natural generalization of the "sliding window" technique to two dimension introduces m as a constant multiple. We suggest that in these circumstances one should try to separate the values computed. This implicit thought process has led to efficient algorithms in the past. We make the concept explicit and believe it will prove useful for solving various two-dimensional problems in the future.
2. We use the separable values method on the submatrices character count problem. This is a basic problem that, surprisingly, did not seem to be solved in the literature. It has turned out to be

deceptively tricky. Separable values enabled us to find an efficient solution. However, the value we ended up using, while simple, is by no means obvious.

This paper is organized as follows. In Section 2 we introduce the notion of *separable convolutions* and show how it can be used to solve the character count problem for fixed alphabets of finite size. In Section 3 we generalize the notion of separable convolutions to separable values that are not convolutions. We introduce the value that will efficiently solve the submatrices character count problem. Section 4 gives the formal definition of the value used. The outline of our algorithm is presented in Section 5. The main technical difficulties of efficiently updating the value are handled in Section 6.

2. Separable convolutions

Convolutions are used for filtering in signal processing and other applications. Two-dimensional convolutions can be more efficiently computed in case of a *separable convolution*. In essence, the convolution is performed twice: once vertically on all columns, and then horizontally on all rows.

A convolution uses two initial functions, T and P, to produce a third function M. We formally define a discrete convolution.

Definition 5. Let T be a real-valued function whose domain is $\{1,\ldots,n\}$ and P a real-valued function whose domain is $\{1,\ldots,m\}$. We may view T and P as arrays of numbers, whose lengths are n and m, respectively. The *discrete convolution of T and P* is the polynomial multiplication

$$M[j] = \sum_{i=1}^{m} T[j+i-1]P[i], \quad j = 1,\ldots,n-m+1.$$

In the general case, the convolution can be computed by using the fast Fourier transform (FFT) [8]. This can be done in time $O(n \log m)$, in a computational model with word size $O(\log m)$. However, for special cases of P, the convolution can be computed in linear time. For example, the subsequence sum problem in Definition 1 is a convolution where $P[i] = 1$, for $i = 1,\ldots,m$. It can be computed in linear time.

In a two-dimensional convolution, both T and P are two-dimensional functions, i.e., T is an $n_1 \times n_2$ matrix and P is an $m_1 \times m_2$ matrix. For ease of notation we will assume that T and P are squares, although all definitions and results in this paper apply to rectangles.

Definition 6. (a) Let T be a function whose domain is $\{[i,j] \mid i,j = 1,\ldots,n\}$ and P a function whose domain is $\{[i,j] \mid i,j = 1,\ldots,m\}$. We may view T and P as square matrices of numbers, whose sizes are n^2 and m^2, respectively. The *two-dimensional discrete convolution of T and P* is

$$M[i,j] = \sum_{\ell=1}^{m} \sum_{k=1}^{m} T[i+k-1, j+\ell-1]P[k,\ell].$$

(b) A convolution is *separable* in the special case where the matrix P, whose size is m^2 can be expressed as the outer product of two vectors of length m: P_{row} and P_{column}, such that: $P[i,j] = P_{\text{row}}[i]P_{\text{column}}[j]$.

The following corollary follows directly from the definitions.

Corollary 1. *If P is separable then the convolution of T and P is*

$$M[i,j] = \sum_{\ell=1}^{m} P_{\text{column}}[\ell] \sum_{k=1}^{m} T[i+k-1, j+\ell-1] P_{\text{row}}[k].$$

The submatrix sum problem in Definition 3(a) is a separable convolution. For every column $T[i,j_0],\ i=1,\ldots,n$, solve the subsequence sum problem and create a new $n \times n$ matrix T' such that $T'[i_0,j_0]$ is the sum of the subsequence $T[i,j_0],\ i=i_0,\ldots,i_0+m-1$. Now solve the subsequence sum problem for every row of T'. The subsequence sum of $T'[i,j]$ is the submatrix sum of $T[i,j]$. Therefore, since the subsequence sum problem can be solved in time $O(n)$ for every column of T and row of T', the submatrices sum problem can be solved in time $O(n^2)$.

Corollary 2. *The submatrices character count problem can be solved in time $O(n^2)$ for matrix T over a finite alphabet Σ.*

Proof. Let $v \in \Sigma$. Define the function $\chi_v : \Sigma \to \{0,1\}$ as follows. Let $s \in \Sigma$. Then

$$\chi_v(s) = \begin{cases} 1 & \text{if } s = v; \\ 0 & \text{otherwise.} \end{cases}$$

Extend the definition to arrays in the natural form, i.e., if T is an $n \times n$ array then $\chi_v(T)$ is the $n \times n$ array whose $[i,j]$ element is $\chi_v(T[i,j])$.

For a fixed $u \in \Sigma$, the submatrices sum problem on $\chi_u(T)$ can be solved in time $O(n^2)$. The crucial observation here is that a submatrix sum is 0 iff u does not appear in that submatrix. Therefore we conclude that u appears in precisely the submatrices whose sum is non-zero.

Repeating the above operation $|\Sigma|$ times, allows us to solve the submatrices character count problem in time $O(|\Sigma|n^2)$, which is linear where Σ a finite alphabet. □

3. The idea behind the algorithm for the submatrix character count problem

3.1. The general idea for two-dimensional problems

Another way of viewing the separation of the convolutions is by remarking that the two-dimensional problem has been somehow projected into one dimension in a manner that preserves the information crucial to the problem.

This type of projection has also been used before. Bird [7] and Baker [6] solved the two-dimensional pattern matching problem by using the algorithm of Aho and Corasick [1] for finding all occurrences of all pattern rows in the text, and then the algorithm of Knuth et al. [12] for finding, in the text columns, the occurrences of the string representing the "flattening" of the pattern rows.

We can describe this process as a *separation of values*, where the vertical and horizontal values need not be convolutions and need not be equal. They only need to satisfy the following conditions.

For a fixed row i, identify a value A, such that $A[i,j]$ encodes some value for the *subcolumn* of length m starting at $T[i,j]$, i.e., $A[i,j]$ encodes a value for $T[i+k,j],\ k=0,\ldots,m-1$. The value A should have the following properties:

1. It is possible to compute the required submatrix problem for $T[i+k,j+\ell],\ k,\ell=0,\ldots,m-1$ from $A[i,1],A[i,2],\ldots,A[i,j+m-1]$ in "close to" $O(n)$ time.
2. Given the computation of the submatrix problem from $A[i,j],A[i,j+1],\ldots,A[i,j+m-1]$, and given $A[i,j+m]$, it is possible to efficiently compute the submatrix problem for $T[i+k,j+1+\ell],\ k,\ell=0,\ldots,m-1$.
3. Given $A[i,\ell],\ \ell=1,\ldots,n$, it is possible to compute $A[i+1,\ell],\ \ell=1,\ldots,n$ in time "close to" linear in n.

Example 1. The separable convolution used for the submatrix sum problem is described in this context quite easily. Simply take $A[i,j]$ to be $\sum_{k=0}^{m-1} T[i+k,j]$. All above properties hold.

1. The submatrix sum at location $[i,j]$ (denoted by $\sigma[i,j]$) is $\sum_{\ell=0}^{m-1} A[i,j+\ell]$.
2. $\sigma[i,j+1]=\sigma[i,j]-A[i,j]+A[i,j+m]$, thus we can compute the submatrix sum at location $[i,j+1]$ in constant time, using the submatrix sum at location $[i,j]$.
3. $A[i+1,j]=A[i,j]-T[i,j]+T[i+m,j]$, thus the values for a new row can be computed in time $O(n)$ using the previous row.

Unfortunately, such a simple scheme does not work for the submatrix character count problem since the appearance of a character in one subcolumn affects whether this character's appearance in another subcolumn should be counted. (In the submatrix sum problem, the sum of each subcolumn was computed independently of the sum of its neighboring subcolumns.) We therefore propose a different scheme for the subsequence character count problem. This new scheme generalizes to two dimensions.

3.2. New scheme for the subsequence character count problem

We will employ for one dimension an idea similar to the projection described above. We will define an array A that enables a linear time computation of the subsequence character count problem. However, since the problem is one-dimensional, A is a vector.

For the first m positions, the value of $A[i]$ is $+1$ if the character in $S[i]$ does not appear previously, otherwise it is 0. Subsequently, value A is $+1$ at a location i if the subsequence $S[i-m+1\ldots i]$ has one more character than the subsequence $S[i-m\ldots i-1]$. It is -1 if the subsequence $S[i-m+1\ldots i]$ has one character fewer than the subsequence $S[i-m\ldots i-1]$. It is 0 if both subsequences have the same number of symbols.

A formal local definition of A is:

For $i=1,\ldots,m$:

$$A[i]=\begin{cases}+1 & \text{if } S[i]\neq S[i-\ell],\ \ell=1,\ldots,i-1,\\ 0 & \text{otherwise.}\end{cases}$$

For other i define

$$A[i] = \begin{cases} +1 & \text{if } S[i] \neq S[i-\ell],\ \ell = 1,\ldots,m \text{ but } \exists\ \ell \in \{1,\ldots,m-1\} \text{ such that } S[i-m] = S[i-\ell], \\ -1 & \text{if } S[i-m] \neq S[i-\ell],\ \ell = 0,\ldots,m-1 \text{ but } \exists\ \ell \in \{0,\ldots,m-1\} \text{ such that } S[i] = S[i-\ell], \\ 0 & \text{otherwise.} \end{cases}$$

Example 2. Let the window size $m = 5$, and let S and A be arrays of length 20. Both S and A are represented below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S	a	b	c	a	d	e	f	c	b	a	g	e	f	a	b	c	a	d	b	e
A	+1	+1	+1	0	+1	+1	0	0	0	0	0	0	0	−1	+1	0	−1	0	0	+1

1. $A[4] = 0$ since $S[4] = a$ appears within the window ($S[1] = a$).
2. $A[6] = +1$ since $S[6] = e$ is new but $S[1] = a$ still appears in the window.
3. $A[7] = 0$ since $S[7] = f$ is new and $S[2] = b$ does not appear in the window.
4. $A[14] = -1$ since $S[14] = a$ appears previously in window ($S[10]$) but $S[9] = b$ departs and does not appear in the window.

It is easy to see that the required properties hold for value A (only the first two are relevant since the problem is one-dimensional):

1. The subsequence character count at location i is $\sum_{j=1}^{i+m-1} A[j]$.
2. There are a number of straightforward ways to implement the efficient computation of $A[i+1]$. We briefly describe one that we later generalize to the submatrix character count problem. $A[i],\ i = 1,\ldots,m$ are computed in a manner similar to the one described in Section 1. For $i > m$, when symbol $S[i]$ is encountered a few actions are taken:
   (a) Write -1 in $A[i+m]$ (that is where the new introduced symbol will "drop off" assuming it does not occurs again between now and then).
   (b) If $S[i] \neq S[i-\ell],\ \ell = 1,\ldots,m-1$ then $A[i] \leftarrow A[i]+1$ (recall that it is possible that there was a -1 in $A[i]$).
   (c) if $\ell \in \{1,\ldots,m-1\},\ S[i-\ell] = S[i]$, and $S[i] \neq S[i-j], \forall j,\ \ell > j > 0$ (i.e., the character $S[i]$ appears another time within the last window of size m, and its last appearance is in $S[i-\ell]$) then do nothing to $A[i]$ but remove the -1 from $A[i-\ell+m]$, i.e., set $A[i-\ell+m] \leftarrow 0$.

   The actions above define the correct value of A. Clearly all operations are constant time except for the verification of $S[i]$'s occurrence in $S[i-\ell],\ \ell = 1,\ldots,m-1$ which can be done (as in Section 1) in time O(1) for finite alphabets, O(log m) for ordered alphabets in the comparison model, and O(m) in unordered alphabets.

3.3. Idea for the submatrix count problem

We generalize this new scheme for the subsequence character count problem to two dimensions. We are now considering an $n \times n$ matrix T. Whereas $A[i]$ before told us if symbol $S[i]$ adds to the character count or the loss of $S[i-m]$ subtracts from it (or both), we now incorporate the effect of subcolumn $T[i, j+\ell],\ \ell = 0,\ldots,m-1$. We will call the value array the *Delta Vector*. Each

row of T has a delta vector. We will refer to an $n \times n$ array D as the *delta vector array*, where the ith row of D is the delta vector of row i of T. The reason we name our value the *delta vector* is that $D[i,j]$ is the difference between the number of new characters introduced by subcolumn $T[i, j+\ell],\ \ell = 0,\ldots,m-1$ and the number of characters that are lost in the submatrix as a result of dropping subcolumn $T[i-m, j+\ell],\ \ell = 0,\ldots,m-1$.

It is clear that the number of different characters in the submatrix whose origin is $[i,j]$ $(T[i+k, j+\ell],\ k,\ell = 0,\ldots,m-1)$ is

$$\sum_{\ell=1}^{j+m-1} D[i,\ell].$$

The remainder of this paper and the main effort herein is devoted to the formal definition and efficient computation of the value – the delta vector $D[i,j]$.

4. Formal value definition

Definition 7. Let T be a $n \times n$ array. A *strip* is an m row submatrix of T, with the first strip being the first m rows, the second strip is rows 2 to $m+1$, and the ith strip is rows i to $m+i-1$.

Note that every two consecutive strips have $m-1$ rows in common. In fact, moving from strip i to strip $i+1$ means "peeling off" row i and adding row $i+m$.

Before formally defining the delta vector, we need the concept of *character effective region.*

Definition 8. The *effective region* of character v in strip i is the set of non-overlapping intervals $E_v^i = \{[a_\ell, b_\ell] \mid 1 \leqslant a_\ell < b_\ell \leqslant n,\ \ell = 1,\ldots,k\}$ such that for all ℓ:

1. v appears in column a_ℓ of the strip.
2. For $b_\ell < n$, v does not appear in columns $b_{\ell-j},\ j = 0,\ldots,m-2$ but does appear in column $b_{\ell-m+1}$.
3. For any $d \in [a_\ell, b_\ell]$ v appears in one of the columns $d-j,\ j = 0,\ldots,m-1$.

In addition, the following conditions hold for the intervals in E_v^i:

1. v does not appear in any column not in the intervals.
2. For every column j such that the symbol v appears in that column, there exists a unique interval $[a_{\ell_0}, b_{\ell_0}] \in E_v^i$ such that $j \in [a_{\ell_0}, b_{\ell_0}]$.
3. No pair of intervals overlap.

The intuitive meaning of the effective region of a character is the area of the strip where that character appears in an $m \times m$ matrix. A character that appears in a strip column, no matter on which row, can affect all the sub matrices that contain this column. Suppose that a character appears first in column j then the effective region of this character will be $[j, j+m-1]$, means that every matrix that ends within the interval contains this character. Now suppose that a new instance of this character appears later, say at column k, $k > j$. If $k > j+m-1$ then k does not belong to the previous interval and it creates a new interval $[k, k+m-1]$. If k belongs to the previous interval $(k \leqslant j+m-1)$ then k will enlarge the previous interval to $[j, k+m-1]$. The set of these intervals is that character effective region.

a	a	b	**c**	b	d	b	d	e	b	a	b	**c**	b	**c**	b	b	a	a	b	a	b
b	a	b	b	b	d	b	b	b	a	a	a	b	**c**	d	e	e	b	b	a	a	**c**
a	b	a	b	a	b	**c**	d	b	b	a	b	b	b	**c**	b	d	a	a	a	b	a
b	a	b	a	b	b	d	e	b	a	b	**c**	b	**c**	d	b	d	b	b	b	a	**c**

Fig. 1. The effective region of c (shaded).

Example 3. Let strip i be:

a a b **c** *b d b d e b a b* **c** *b* **c** *b b a a b a b*
b a b b b d b b b a a a b **c** *d e e b b a a* **c**
a b a b a b **c** *d b b a b b b* **c** *b d a a a b a*
b a b a b b d e b a b **c** *b* **c** *d b d b b b a* **c**

Let the symbol be c (boldface in the example) and let the window size be 4. Then
$E_c^i = \{[4,10],[12,18],[22,22]\}$. A schematic of the effective region can be seen in Fig. 1.
$E_a^i = \{[1,8],[10,15],[18,22]\}$.
$E_b^i = \{[1,22]\}$.
$E_d^i = \{[6,11],[15,20]\}$.
$E_e^i = \{[8,12],[16,20]\}$.

We are now ready to formally define the *delta vector* for strip i.

Definition 9. $D_i[j] = |\{v \mid \exists \ell\ [a_\ell, b_\ell] \in E_v^i \text{ and } j = a_\ell\}| - |\{v \mid \exists \ell\ [a_\ell, b_\ell] \in E_v^i \text{ and } j - 1 = b_\ell\}|$.

In words, the difference between the number of effective region intervals that begin in column j and the effective region intervals that end in column $j-1$ of the strip. This actually denotes the difference between the number of new characters in the window and the characters that are departing from the window.

Notation: For simplification, when the strip is fixed, we will drop the index and denote D_i by D.

Example 4. The delta vector for the strip in Example 3 is:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
D	+2	0	0	+1	0	+1	0	+1	−1	+1	−1	0	−1	0	+1	0	0	+1	−1	0	−2	+1

Note the following:
1. $D[4] = +1$ because an interval of the effective region of c begins at column 4 and no interval finishes. This means that c is a new character introduced to the window, and no character is dropped.
2. $D[6] = +1$ because the first interval of E_d^i begins and no interval concludes. This means that d is a new character but no character was dropped from the window (the characters in column 2 are a and b, both appear in columns 3–5).

3. $D[9] = -1$ since the first interval of E_a^i concludes at column 8 and no new interval begins at column 9. This mean that a, that appears in column 5 but does not appear in columns 6, 7 or 8 dropped from the window. The characters in column 9 (e and b) previously appear in the window.
4. $D[11] = -1$ because the first interval of E_c^i ends in column 10 and no new interval starts.
5. $D[12] = 0$ because the first interval of E_d^i ends in column 11 and the second interval of E_c^i begins in column 12. This means that of the characters in column 8 (d, e and b), both e and b appear in the window but the d is dropped. On the other hand, of the characters of column 12 (a, b and c), both a and b are not new to the window but c is new.
6. $D[21] = -2$ because both second intervals of E_d^i and E_e^i end in column 20 but column 21 does not have any starting intervals (does not introduce new characters to the window).

As mentioned in Section 3.3, for the $m \times m$ submatrix whose top leftmost corner is in location $[i, j]$, the number of different characters is:

$$\sum_{\ell=1}^{j+m-1} D[i, \ell].$$

This can clearly be computed in linear time for every row.

Example 5. The array denoting the prefix sums of D in Example 4 is:

2 2 2 3 3 4 4 5 4 5 4 4 3 3 4 4 4 5 4 4 2 3.

It is easy to verify that this indeed is the number of distinct characters in every location.

Our remaining task is to efficiently compute the delta vector D. We distinguish between constructing the delta vector for strip 1 and updating the delta vector for strip i to become the delta vector for strip $i + 1$.

5. Algorithm outline

5.1. Constructing the delta vector for strip 1

The delta vector is constructed considering the effective regions of all the characters. For each interval $[j_1, j_2]$ that belongs to one of the character's effective region intervals, the delta vector is updated. Conceptually, there are two necessary steps per update:

1. increment the delta vector at index j_1 ($D[j_1] \leftarrow D[j_1] + 1$).
2. decrement at index $j_2 + 1$ ($D[j_2 + 1] \leftarrow D[j_2 + 1] - 1$).

The delta vector is the result of the above updates for all effective regions of all characters.

In reality, we will not be maintaining an effective regions data structure. Rather, the delta vector is updated as symbols are added to or deleted from the strip. The algorithm idea is to move column by column in the strip. For each column go over all the different characters. For each character recalculate the character's effective region and update the delta vector accordingly.

Efficient construction of the delta vector requires the usage and maintenance of some auxiliary variables and data structures. These auxiliary data structures are described in detail in the next section.

```
Algorithm outline – Delta vector construction for first strip
  For j = 1 to n do
    For each different character c in column j
      If c appears in the last m − 1 column then
        We are already in one interval of the effective region of c.
        Update D - Enlarge this region.
      Else
        Update D - Create a new effective region.
      Endif
    End for each
  Endfor j
end Algorithm
```

In the next section we will show that the delta vector for the first strip and all the auxiliary variables can be constructed in time $O(mn)$.

5.2. Update delta vector for next strip

The delta vector for the $(i+1)$st strip is constructed by updating the delta vector of the ith strip. Moving to the next strip means that row i of the matrix T is replaced by row $i+m$. From the column's point of view, for each column j, $j = 1, \ldots, n$, the old character $T[i, j]$ is replaced by a new character $T[i+m, j]$. This replacement may generate two changes to the delta vector. If the new character did not appear in this column before and if the old character is the last instance of the character in this column (including the new character). In these cases we have to recalculate the effective regions of the old and new characters and update the delta vector accordingly.

```
Algorithm outline – Delta vector strip update
  For j = 1 to n do
    If new character didn't appear before in this column then
      Recalculate new character effective region and update D
    If old character doesn't appear any more in this column then
      Recalculate old character effective region and update D
  Endfor j
end Algorithm
```

In the next section we will show that we can build the next strip delta vector and update all its auxiliary variables in time of $O(n \log m)$.

Total algorithm time: There are $n - m + 1$ strips in matrix T. The delta vector for the first strip is constructed in time $O(nm)$. Subsequently, we update each of the $n - m$ remaining strips in time $O(n \log m)$. The total is, then $O(mn + (n-m)n \log m) = O(n^2 \log m)$.

6. Algorithm details

The efficiency of our algorithm lies in the auxiliary data structures. This section presents the definition of the data structures and their maintenance. For simplicity's sake we assume a finite alphabet. We later mention how to generalize to alphabets of unbounded size.

6.1. The auxiliary data structures

The data structures below are initialized for the first strip and updated as we move from strip to strip.

1. $V_c[j]$: the number of instances of char c in column j.
   Purpose: This variable helps us keep track of whether a new character is introduced or dropped from a column as the strip is moved.
   Maximum Size: For finite alphabet $O(n)$ since there are at most a constant number of characters per column. If the alphabet is infinite, there may be n^2 different characters in T. However, in any strip there are at most $mn+1$ different characters active. Therefore this data structure can be implemented in space $O(nm)$ with $O(\log m)$ access time (by keeping the columns' characters in balanced search trees (see, e.g., [8])).
2. H_c: the number of columns, among the $m-1$ columns to the left of the current active column, that character c appears in.
   Purpose: Utilized in building the first delta vector. Helps to determine whether c enlarges an existing effective interval or creates a new one.
   Maximum Size: For finite alphabet constant space and access time. For infinite alphabet bounded by $O(m^2)$ space with a $O(\log m)$ access time.
3. L_c: a list of all the columns in which c appears.
   Purpose: Help recalculate the character effective region and update D accordingly.
   Implementation: The list is implemented in a data structure that supports the operations *insert, delete, predecessor, successor* and *last*. For example, for each character c, L_c may be maintained as a Red-Black tree. This enables *insert, delete, predecessor* and *successor* in time $O(\log n)$ and *get last* in constant time.
   Maximum Size: As in $V_c[j]$ the required space is $O(n)$ for a finite alphabet and $O(mn)$ for infinite alphabet.

Example 6. In the strip from Example 3 the $V_c[j]$ and L_c values are as depicted below. We only wrote the non-zero $V_c[j]$.

$V_c[4]=1, V_c[7]=1, V_c[12]=1, V_c[13]=1, V_c[14]=2, V_c[15]=2, V_c[22]=2.$
$L_c = 4, 7, 12, 13, 14, 15, 22.$

6.2. Algorithm for first delta vector

The first delta vector and its auxiliary data structures can be constructed in a fairly straightforward manner in time $O(nm)$ for finite alphabets. Simply go down the columns of the strip starting from the leftmost column. For every symbol, check if it is the first appearance in the last size m

window, in which case the appropriate variables are updated. For repeating occurrences, update H, V, and L appropriately.

A pseudo-code of the algorithm appears below.

Algorithm for First Strip

```
Initialize H, V, L and D.
For j =1 to n do /* For each column of the first strip */
    For i=1 to m do /* For each character in current column */
        c ← T[i, j]
        /* If first instance of this character in the last m columns of strip */
        If H_c = 0 do
            D[j] ← D[j] + 1 /* character is effective from the first instance column */
        /* Elseif first instance of character c in curr column (but not in the last m columns */
        Elseif V_c[j] = 0 do
            ℓ ← last(L_c) /* col of prev instance of character c */
            D[ℓ + m] ← D[ℓ + m] + 1 /* This character will be effective only after */
            /* effectiveness of prev instance */
        Endif
        V_c[j] ← V_c[j] + 1
        /* If first instance of character c in curr column */
        If V_c[j] = 1 do
            D[j + m] ← D[j + m] − 1 /* character c is effective until column j + m */
            H_c ← H_c + 1
            insert(L_c, j)
        Endif
    Endfor i
    /* Update H */
    If j > m do
        ℓ ← j − m
        For i = 1 to m do /* For each character in column j − m */
            d ← T[ℓ, i]
            V_d[ℓ] ← V_d[ℓ] − 1
            If V_d[ℓ] = 0 do H_d ← H_d − 1
        Endfor i
    Endif
Endfor j
```

end Algorithm

6.3. Updating the delta vector

Equipped with $V_c[j]$, L_c and D of the ith strip, we are ready to update those variables for the $(i+1)$st strip. Moving to the next strip means that row i is dropped and row $i+m$ is added. From the column's point of view, for each $j, j = 1, \ldots, n$, the character $T[i, j]$ is dropped and the character $T[i+m, j]$ is added. If the new character does not appear in its column yet then we have to update D

with the character effective region. The update is performed by first, inserting the index j to $L_{\text{new\_}c}$, the column list of the new character. Second, calculating the new effective region considering the previous column and next column of the new character appearance. We will deal with the old character in a similar way. If it is the only instance of this character in the column (includes the new character) then we have to delete the index j from $L_{\text{old\_}c}$ and update D considering the previous column and next column of the old character appearance.

D is updated using V and L in the following manner. Let $T[i+m, j] = c$ and assume $V_c[j] \neq 0$. Clearly we must increment $V_c[j]$ by 1. Nothing else needs to be changed, since there is no new effect by $T[i+m, j] = c$.

However, if $V_c[j] = 0$ then the introduction of c in column j needs to be noted. $V_c[j]$ is incremented by 1, and j is inserted into L_c. We need to compute the effects on D.

These effects are dependent on the index immediately preceding j in list L_c, denoted by p, and the index immediately following j in L_c, denoted by s. We will say that index k is *far from* index ℓ if they are not within the boundaries of the same submatrix, i.e., $|k - \ell| \geqslant m$.

The following four cases are possible.

1. j is far from both p and s. This means that we need to add a new interval $[j, j+m-1]$ to the effective region of c. The implementation of this action is incrementing $D[j]$ by 1 and decrementing $D[j+m]$ by 1.
2. j is far from p but not far from s. This means that the interval including p ends before j, thus is unchanged. But the interval starting at s now needs to start at j. Thus we decrement $D[s]$ by 1 and increment $D[j]$ by 1.
3. j is far from s but not far from p. The closeness to p means that the interval including p will not end at $p+m-1$ but rather will extend to $j+m-1$. This is implemented by incrementing $D[p+m]$ and decrementing $D[j+m]$. However, the interval starting at s is unchanged.
4. If j is not far from either p or s there are two possibilities.
   (a) p and s are themselves not far from each other. In this case nothing needs to be done since j does not change the effective region interval at all.
   (b) p and s are far from each other. This means that the interval of p ends at $p+m-1$ which is before the start of the interval that starts at s. Adding c at j unifies those intervals into a single interval that includes also the entire area between p and s. Thus we increment $D[p+m]$ and decrement $D[s]$.

The above cases precisely define the necessary updates to D resulting from the addition of $T[i+m, j]$. The changes to D resulting from the deletion of $T[i, j]$ are complementary. Simply exchange every "increment" and "decrement" in the above cases.

Example 7. Consider strip i from Example 3, and assume the next row is as in Fig. 2.

1. The first element being deleted is a (column 1) and then c is added. This changes $V_c[1]$ from 0 to 1. L_c changes from $4, 7, 12, 13, 14, 15, 22$ to $1, 4, 7, 12, 13, 14, 15, 22$.
   This means that we are in case 2, far from p and close to s. Thus $D[1]$ is incremented from 2 to 3. $D[4]$ is decremented from 1 to 0.
2. The next change is in column 4 where c is dropped from the top line and b is added to the bottom line. $V_c[4]$ changes from 1 to 0. L_c changes from $1, 4, 7, 12, 13, 14, 15, 22$ to $1, 7, 12, 13, 14, 15, 22$.
   This is the complement of case 4(b). Thus $D[7]$ is incremented to 1 and $D[5]$ is decremented to -1.

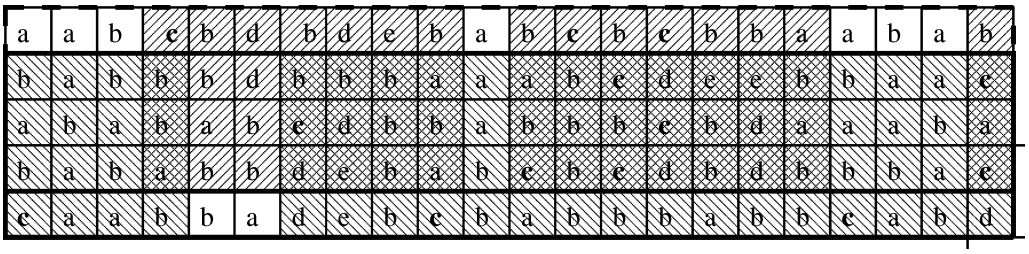

Fig. 2. The changes in the effective region of c (shaded).

Using the cases described above to encode "update D", the pseudo-code for updating D at a change of strips is:

Algorithm for Updating *D* at Change of Strip

For $j = 1$ to n do /* For each column of the strip */
  /* Add $T[i+m, j]$ */
  $c \leftarrow T[i+m, j]$
  $V_c[j] \leftarrow V_c[j] + 1$
  If $V_c[j] = 1$ do /* First instance of c in this column */
    insert(L_c, j)
    prev_col $\leftarrow$ *predecessor*(L_c, j)
    next_col $\leftarrow$ *successor*(L_c, j)
    Update D
  Endif
  /* Remove $T[i, j]$ */
  $c \leftarrow T[i, j]$
  $V_c[j] \leftarrow V_c[j] - 1$
  If $V_c[j] = 0$ do /* Only instance of c in this column */
    delete(L_c, j) /* L_c must have column j */
    prev_col $\leftarrow$ *predecessor*(L_c, j)
    next_col $\leftarrow$ *successor*(L_c, j)
    Update D
  Endif
Endfor j

end Algorithm

Time for updating *D* at strip change: This algorithm goes over all the columns and for each column it calculates the functions *insert, delete, predecessor* and *successor* a constant number of times. These functions can be implemented in time $O(\log n)$ and this makes the total time for updating D at every strip change $O(n \log n)$.

6.4. Reducing the log n to log m

Note that the height of a strip is m, thus the $\log n$ factor is a result of the length of the strip. If the strip lengths were $2m$ then our time would be $O(n \log m)$. However, we can easily assume that

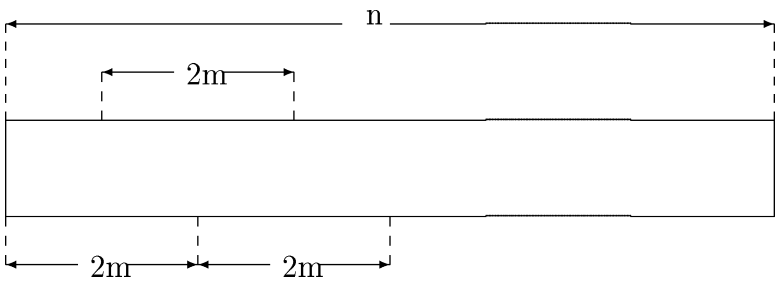

Fig. 3. Dividing the text into overlapping segments.

the strip lengths are $2m$ because of the following observation that is commonly used in pattern matching papers (e.g., [2,4]).

Observation 1. *If the submatrices character count problem can be solved in time $f(m)$ for strips of length $n \leqslant 2m$, then it can be solved in time $\frac{n}{m} f(m)$ for any n-length strip. Simply divide the text into $\frac{n}{m}$ overlapping $2m$-length segments (see Fig. 3) and solve the submatrices character count problem separately for each. Clearly, all locations are covered.*

6.5. Infinite alphabet size

Note that our time of $O(n \log m)$ for update of D after strip change is necessary even for finite alphabet because of our need to maintain the L list's operations. However, the V structure can be maintained in linear time. An infinite alphabet size will cause maintaining the V data structure to also cost $\log m$ per operation, however, this will not change the total algorithm's time.

7. Conclusions

The submatrices character count problem seems to be a simple one yet its solution was not trivial. We found that an efficient solution was finally derived after we separated the sought value to horizontal and vertical values, in the spirit of the engineering concept of separable convolutions. This thought process was implicitly used in a number of problems in the past and we recommend keeping it explicitly in mind when considering subarray problems in two-dimensional arrays.

Our treatment of the topic was purely theoretical. It would be interesting to experimentally calculate the value of m for which it becomes practical to use the separable values algorithm as opposed to the trivial $O(n^2 m)$ sliding windows algorithm.

Acknowledgment

The authors thank S. Muthukrishnan for pointing out the work on color range queries.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching, Commun. ACM 18 (6) (1975) 333–340.
[2] A. Amir, Y. Aumann, G. Landau, M. Lewenstein, N. Lewenstein, Pattern matching with swaps, J. Algorithms 37 (2000) 247–266 (Preliminary version appeared at FOCS 97).
[3] A. Amir, K.W. Church, E. Dar, Separable attributes: a technique for solving the submatrices character count problem, in: Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 400–401.
[4] A. Amir, M. Farach, Efficient 2-dimensional approximate matching of half-rectangular figures, Inform. Comput. 118 (1) (1995) 1–11.
[5] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inform. Process. Lett. 49 (1994) 111–115.
[6] T.J. Baker, A technique for extending rapid exact-match string matching to arrays of more than one dimension, SIAM J. Comput. 7 (1978) 533–541.
[7] R.S. Bird, Two dimensional pattern matching, Inform. Process. Lett. 6 (5) (1977) 168–170.
[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, McGraw-Hill, 1992.
[9] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics: Principles and Practice, Addison-Wesley, Reading, MA, 1990.
[10] J. Hafner, H.S. Sawhney, W. Equitz, M. Flickner, W. Niblack, Efficient color histogram indexing for quadratic form distance functions, IEEE Trans. Pattern Anal. Mach. Intell. 17 (7) (1995) 729–736.
[11] R. Janardan, M.A. Lopez, Generalized intersection searching problems, Int. J. Comput. Geom. Appl. 3 (1993) 39–69.
[12] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.
[13] A.C.C. Yao, Near-optimal time-space tradeoff for element distinctness, in: Proceedings of the 29th IEEE Annual Symposium on Foundations of Computer Science, 1988, pp. 91–97.